Globe: A Wide-Area Distributed System

van Steen, M.R.; Homburg, P.C.; Tanenbaum, A.S.

published in
IEEE Concurrency
1999

DOI (link to publisher)
10.1109/4434.749137

document version
Publisher's PDF, also known as Version of record

Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum
Vrije University

Globe: A Wide-Area Distributed System

The authors present an object-based framework for developing wide-area distributed applications. The World Wide Web's current performance problems illustrate the benefit of encapsulating state, operations, and implementation strategies on a per-object basis. The authors describe how distributed objects can implement worldwide scalable Web documents.

The Internet's spectacular growth has the potential to connect a billion computers into an integrated distributed system that offers numerous applications for science, commerce, education, and entertainment. The hardware and communications infrastructure needed is rapidly being deployed, but the software infrastructure is still lacking. We propose Globe: a novel scalable infrastructure for a massive worldwide distributed system.

Currently, designers build applications on top of a limited number of communication services. On the Internet, for example, this means that applications communicate mainly through transmission-control protocol (TCP) connections, but otherwise must implement all additional services themselves, including services for naming, replication, migration, fault tolerance, and security.

As a further example, consider the World Wide Web (see the sidebar on World Wide Web documents). The Web implements its own communication protocol, hypertext-transfer protocol (HTTP), on top of TCP. It uses a tailor-made naming system based on uniform resource locators (URLs). Replication is supported in the form of caches that are part of Web proxies—they cannot be used for other applications (cache-coherence protocols rely on Web-page attribute fields). Unfortunately, few measures have been taken to handle broken links and server crashes. Although security has been proposed in the form of a universal HTTP extension, proprietary solutions such as SSL from Netscape are taking shape. Other Internet applications such as e-mail and Usenet news each have their own software models and infrastructure, but no commonality exists among any of them.

Consequently, building new wide-area applications is difficult. First, too much effort is repeatedly spent on implementing common or standard services that should have already been there. Second, with application-specific services, interoperability between different applications can be difficult or even impossible.

Instead, we propose a different approach. Rather than developing applications directly on top of the transport layer, we want a software infrastructure that provides us with a set of common distribution services. The main requirement is that this infrastructure, or *middleware*, can scale to support a billion users all over the world.

Scalable middleware requirements

Our solution lies in the development of a wide-area distributed system called Globe (visit *http://www.cs.vu.nl/~steen/globe/*). We aim to meet three major design objectives: provide a uniform model for distributed computing, support a flexible implemen-

1092-3063/99/$10.00 © 1999 IEEE

An example: scalable World Wide Web documents

To illustrate the benefits of our approach, we consider how Globe facilitates support for scalable Web documents. A Web document is taken to be a collection of logically related pages, including their icons, sounds, applets, and so on.

Proposals for caching or replicating Web documents tend to treat pages alike in the sense that the semantics of a document are not taken into account. Documents and their pages are treated differently only by considering metadata such as access statistics, times of modification, and other relevant information. Alternatively, some solutions are tailored to a specific class of documents and are not universally applicable.

As Web documents are becoming more diverse, it is clear that it will be hard to find a single solution to use in all situations. For example, current caching and replication schemes for the Web assume that pages are modified at only one location. They are not suited to support Web pages several users actively share, such as shared whiteboards and pages manipulated through groupware editors. Likewise, it is hard to tailor a replication scheme to just a single document, as is needed with mail-distribution lists.

The approach followed in Globe is radically different. Rather than searching for generally applicable replication schemes, each distributed object can adopt its own strategy. Globe offers a library of different replication subobjects (see Figure 2 in the main text) that can be adopted and subsequently fine-tuned separately for each distributed object. When required, new ones can be constructed.

For example, consider the current major application of the Web, namely providing information through a logical Web site, also called a home page. A home page is related to a person, project, consortium, or organization, and is generally the entry point of an entire hypertext document consisting of multiple pages. Typically, in Globe, this document would be modeled as one distributed shared object. The state of such a document consists of the rooted directed graph of individual pages that make up the hypertext document.

Web sites can be very different with respect to the kind of documents they manage. Pages of a personal site rarely require any replication and possibly only short-lived caching. In Globe, the owner of a personal site would group the pages into a single document and provide only a single contact address. When a user binds to that document, its pages (including icons, images, text, and graphics) are transferred to the user's browser, possibly in parts as in current practice, and are subsequently written to the user's private cache. Note that there is generally no need to write pages of such a document to a site-wide cache as Web proxies do.

On the other hand, an organization's Web site might be of an entirely different nature. First, we might assume that its popularity is much higher than that of personal Web sites. Also, in the case of multinational organizations, readers will come from all over the world. In these cases, a primary-backup approach where pages are replicated to a number of mirror sites is useful. The organization's Web site could be constructed as one or more Web documents, where each document is registered at the location service with multiple contact addresses. The nearest address is always returned to a user. Note that, in Globe, the name of a Web document can be the same everywhere. Also, there is no need to tell the user that there are mirror sites and where these sites are. In contrast to personal Web documents, site-wide caching, as is done by current Web proxies, might now be useful.

There are also Web sites whose content changes rapidly and which might require active replication schemes. For example, Web documents of online news providers might want to use a publish/subscribe type of replication by which subscribers to a provider's document are notified when news updates occur. This also holds for Web documents related to conferences and other types of timely events. In the current Internet infrastructure, mailing lists can automatically propagate updates to users. Such lists are highly inefficient. In Globe, notification would be an integral part of the Web document, using a multicasting scheme appropriate for that document. Of course, notification could be combined with actively replicating the updates, but this might not be appropriate in all cases.

What we see here are similar Web documents that require very different replication strategies. Personal home pages need not be replicated and should be cached on a per-user basis. Organizational home pages can apply primary-backup replication and should be cached per site. Home pages related to timely events might benefit from a publish-and-subscribe type of replication where clients are notified when updates occur. Unfortunately, such distinctions are presently impossible to make. In Globe, however, each Web document can use a replication strategy tailored to its own characteristics.

tation framework, and ensure worldwide scalability.

Uniform model

A distributed system should provide a consistent and uniform view of how to organize applications built on top of it. DCOM and DCE, for example, support client-server computing using only remote procedure calls (RPCs).[1,2] CORBA provides a remote-object model for all its applications.[3] Applications built on top of AFS are offered a wide-area file system based on location-transparent naming.[4] The Web offers a model of worldwide distributed documents tied together through hyperlinks.

A uniform model contributes to a single-system view. In addition, it should integrate common services such as communication, naming, replication, and so on. Moreover, these services should be included in such a way that all aspects related to the distribution of data, computations, and coordination are effectively hidden from users. In other words, a model should provide *distribution transparency*. Worldwide systems that integrate common services and support all types of distribution transparency do not currently exist.

More importantly, distribution services at present generally have a single wired-in general-purpose policy. For example, all proxy caches in the Web work the same way. The same holds for caching in AFS. In CORBA and DCE, client proxies are always the same: they forward requests and handle replies.

Related work

There is much academic and industrial activity on the design and implementation of shared data and objects. A shared-data model offers a small set of primitives for reading and writing bytes to shared regions of storage. Typical examples of shared-data models are network file systems and distributed shared memory implementations. The main problem is achieving performance and scalability while keeping data consistent. DSM and storage systems such as Munin and Khazana follow an approach similar to Globe by attaching replication policies on a per-region basis.[1,2] In most DSM systems, performance is improved by relaxing memory consistency.[3] The main drawback of the shared-data model is that it simply does not provide the level of abstraction needed for developing distributed applications. Therefore, much attention is being paid to object-based approaches.

Objects come with an architectural model that lends itself well to distributed systems. An object can be seen as a fine-grained service provider. To most developers, this means that an object is naturally implemented through its own server process, which handles requests from clients. This view leads to the remote-object model in which a remote-method invocation is made transparent using RPC-like techniques, as is done in DCOM.[4] However, this approach is the major obstacle to scale worldwide. The problem is that remote-object invocations cannot adequately deal with network latencies. Additional mechanisms such as object replication and asynchronous method invocations are therefore necessary.

In the Legion system, objects are located in different address spaces, and method invocation is implemented nontransparently through message passing.[5] The Legion approach is one of the few that explicitly addresses wide-area scalability. The Globus project has developed global pointers to support flexible implementations.[6] A global pointer is a reference to a remote compute object. The pointer identifies a number of protocols to communicate with the object, of which one is to be selected by the client. Global pointers offer a higher degree of flexibility than the Legion approach.

When it comes to distribution transparency, Legion and Globus fall short. Object request brokers explicitly address transparency. An ORB is a mediator between objects and their clients. Basic ORBs provide only support for language-independent and location-transparent method invocation. CORBA-compliant ORBs offer additional distribution services such as naming, persistence, transactions, and so on.[7] Unfortunately, CORBA has not yet defined services for transparently replicating objects or for keeping replicas consistent.

When an ORB is responsible for distribution services, we require additional mechanisms independent of the core object model. One such mechanism is the subcontract used in the Spring system.[8] A subcontract implements an invocation protocol: it describes the effect of a method invocation at the client side in terms of the method invocation at the object's side. For example, in the case of replication, method invocation by a client might result in the invocation of that method

There is no straightforward way to build more sophisticated proxies.

We need mechanisms for implementing object-specific policies (see the "Related work" sidebar). An object should entirely encapsulate such policies. In Globe, we tackle these problems by providing a model of *distributed shared objects*. The two main distinctions with existing models are that our objects can be physically distributed, and each object fully encapsulates its own policy for replication, migration, and so on. In Globe, an object is completely self-contained, so that objects for different applications can have replication (and sometimes policies) carefully tailored to their needs. Nevertheless, all implementation aspects are hidden behind its interfaces to achieve distribution transparency.

Flexible implementation framework

The heterogeneity inherent to a wide-area system should preferably be transparent to applications. However, complete transparency is not always a good idea. For example, we might want to make use of a parallel computer in some computations, so where we do the computation matters. A wide-area distributed system should thus make specialized facilities available to applications when needed. For similar reasons, aspects of the underlying network should be made visible—when bandwidth is scarce, it might be better to move data and computations from server to client, as in the case of Java applets.

What we need is a flexible implementation framework: a set of cooperating mechanisms that make up a reusable design for wide-area distributed applications.[5] It is here that an object-based approach will help. By strictly separating an object's interface from its implementation, we can construct reusable designs by considering only interfaces. We can tailor design toward a specific application by choosing the appropriate object implementations and, where necessary, extending the design with other objects.[6] This is the approach we followed in Globe.

Worldwide scalability

A worldwide-scalable distributed system is capable of offering adequate performance in the face of high network latencies, congestion, overloaded servers, limited resource capacity, unreliable communication, and so on. To achieve worldwide scalability, we at least need to provide extensive support for partitioning and replicating objects.[7]

Adequate support for scaling techniques is precisely what current middleware lacks. DCOM, DCE, and CORBA do not provide the tools for replicating objects. In those cases where caching or replication is supported, such as in AFS and the Web, policies are fixed. However, we can find efficient solutions that scale worldwide only if we take application-level consistency into account. Again, this calls for flexibility.

The Globe system

Globe is a wide-area distributed system that we constructed as a middleware layer on top of the Internet, various Unix systems, and Windows NT. We recently finished our initial architectural design, which consists of an object model and a collection of basic support services. The object model allows for the construction of worldwide scalable objects that a vast number of processes can share. Support

at each replica. Replicating the invocation is encapsulated in the subcontract and hidden from the client. As a general mechanism, subcontracts are too limited. For example, it is hard to develop subcontracts that keep a group of objects consistent for several clients to share.

An alternative approach is to fully encapsulate distribution in an object, leading to a model of partitioned objects. Partitioned objects appeared in SOS in the form of fragmented objects.[9] Globe's distributed shared objects form another implementation of partitioned objects, and have been derived from the Orca programming language.[10]

Fragmented objects in SOS are mostly language-independent. Distribution is achieved manually by allowing interfaces to act as object references that can be freely copied between different address spaces. An important difference with Globe's distributed shared objects is that fragmented objects make use of relative object references. In contrast, Globe's object handles are absolute and globally unique. Fragmented objects have not been designed for wide-area networks. For example, the communication objects have been designed and implemented for local-area networks only. Furthermore, the model does not provide facilities for implementing different coherence policies, nor does it address the problem of platform heterogeneity.

References

1. J. Carter, J. Bennett, and W. Zwaenepoel, "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems," *ACM Trans. Computer Systems*, Vol. 13, No. 3, Aug. 1995, pp. 205–244.
2. J. Carter, A. Ranganathan, and S. Susarla, "Khazana: An Infrastructure for Building Distributed Services," *Proc. 18th Int'l Conf. Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, Calif., May 1998, pp. 562–571.
3. J. Protić, M. Tomačević, and V. Milutinović, "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel & Distributed Technology*, Vol. 4, No. 2, Summer 1996, pp. 63–79.
4. *DCOM Technical Overview*, Microsoft Corp., Redmond, Wash., 1996.
5. A. Grimshaw and W. Wulf, "The Legion Vision of a Worldwide Virtual Computer," *Comm. ACM*, Vol. 40, No. 1, Jan. 1997, pp. 39–45.
6. I. Foster et al., "Managing Multiple Communication Methods in High-Performance Networked Computing Systems," *J. Par. Distr. Comput.*, Vol. 40, No. 1, Jan. 1997, pp. 35–48.
7. *The Common Object Request Broker: Architecture and Specification, Revision 2.2,* OMG Document 96.03.04, Object Management Group, Framingham, Mass., 1998.
8. G. Hamilton, M. Powell, and J. Mitchell, "Subcontract: A Flexible Base for Distributed Programming," *Proc. 14th Symp. Operating System Principles*, ACM Press, New York, 1993.
9. M. Shapiro et al., "SOS: An Object Oriented Operating System—Assessment and Perspectives," *Computing Systems*, Vol. 2, No. 4, Fall 1989, pp. 287– 337.
10. H. Bal and A. Tanenbaum, "Distributed Programming with Shared Data," *Proc. Int'l Conf. Computer Languages*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 82–91.

services include services for naming and locating objects.

Globe's object model

In Globe, processes interact and communicate through distributed shared objects. Each object offers one or more interfaces, with each interface consisting of a set of methods. A Globe object is physically distributed, meaning that its state might be partitioned and replicated across multiple machines at the same time. However, processes are not aware of this: the object completely encapsulates the state and operations on that state. All implementation aspects, including communication protocols, replication strategies, and state distribution and migration, are part of the object and are hidden behind its interface.

For a thread in a process to invoke an object's method, it must first bind to that object by contacting it at one of the object's contact points. A contact address describes such a point, specifying a network address and a protocol through which the binding can take place. (Binding results in an interface belonging to the object being placed in the client's address space, along with an implementation of that interface.) Such an implementation is called a *local object*. Figure 1 illustrates this model and shows a Globe object distributed across four address spaces.

A distributed object is built from local objects, which reside in a single address space and communicate with local objects in other address spaces. They form a particular implementation of a distributed

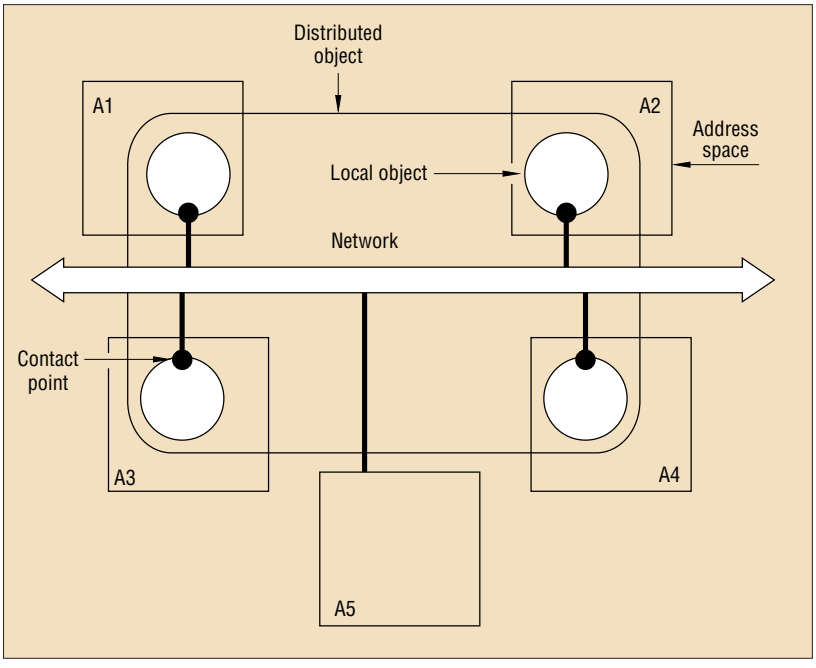

Figure 1. Example of an object distributed across four address spaces. A5 is an additional address space.

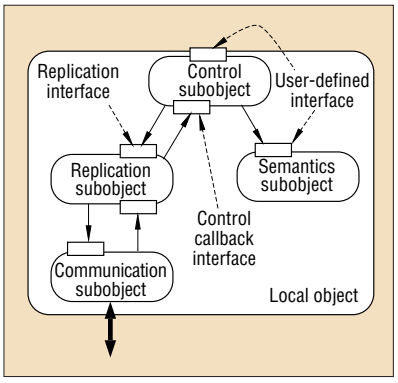

Figure 2. A distributed object's general implementation.

Table 1. Replication interface of a subobject as used by the control subobject.

Method	Description
`start`	Called to synchronize replicas of the semantics subobjects, obtain locks if necessary, and so on.
`invoked`	Called after the control subobject has invoked a specific method at the semantics subobject.
`send`	Provides marshalled arguments of a specific method and passes invocation to local objects in other address spaces.
`finish`	Called to synchronize the replicas again, release locks, and so on.

object's interface. For example, a local object might implement an interface by forwarding all method invocations, as in RPC client stubs. A local object in another address space might implement that same interface through operations on a replica of the object's state.

Our aim is to let application developers concentrate on designing and implementing functionality in terms of objects. Distribution is a different concern, and should be treated separately. For this reason, local objects are constructed in a modular way, to separate issues such as replication and communication from what the object actually does (such as its semantics). We distinguish the following four subobjects, as shown in Figure 2:

- A *semantics subobject* containing the methods that implement the distributed shared object's functionality,
- A *communication subobject* for sending and receiving messages from other local objects,
- A *replication subobject* containing the implementation of a specific replication policy, and
- A *control subobject* handling the control flow within the local object.

These four subobjects are designed to build scalable distributed shared objects. Of course, we also need support for security, persistence, and other services, which are handled by separate subobjects in our approach. As scalability is the focus of this article, we discuss only the four subobjects listed here.

Semantics subobject

The semantics subobject is comparable to objects in middleware such as DCOM and CORBA. It implements (part of) the same functionality the distributed shared object has, thereby ignoring distribution issues. In Globe, a semantics subobject can be implemented in any language; its methods are made available by one or more interfaces. We expect that each subobject implements the standard object interface, which has a role similar to the `IUnknown` interface in COM. Like `IUnknown`, the standard object interface provides a method `getInterface` that returns a pointer to a specified interface.

In principle, the semantics subobjects are the only subobjects a developer must construct. All other parts can either be obtained from libraries or are generated from interface and object descriptions. The only restriction we currently impose is that a control thread is not allowed to block inside a semantics subobject. Instead, a method should return indicating that a condition did not hold. In that case, the control subobject will block the invoking thread, as we explain shortly.

Replication subobject

The distributed object's global state is made up of the state of its various semantics subobjects. In our approach, replication and caching of the semantics subobjects are important techniques for scalability. However, having several copies leads to a consistency problem: changes to one copy make it different from the others. To what extent such inconsistencies can be tolerated depends on the distributed object and the way it is used. Consequently, we need to support coherence protocols on a per-object basis. The replication subobject acts as a placeholder for different protocols and a variety of protocol implementations.

Our basic assumption is that coherence protocols can be expressed in terms of when specific methods of a local semantics subobject copy can be invoked. The replication subobject thus decides when local invocations can take place. Omitting specific details, it offers an interface to the control subobject as shown in Table 1.

In principle, all invocation requests, whether they come from the local client or from the network, are first passed to the replication subobject before the method is invoked at the semantics subobject. When the control subobject receives an invocation request from the local client, it first calls `start` to allow the replication subobject to synchronize the semantics subobject copies. For example, the coherence protocol might require a token to be acquired before any method invocation at the semantics subobject takes place.

The start method returns a set of actions that the control subobject should take. The return value `invoke` tells the control subobject to invoke the method at the semantics subobject. Likewise, `send` instructs the control subobject to pass the marshalled invocation arguments to the replication subobject by subsequently calling `send`. So, for example, with a replication strategy where a method has to be invoked at all replicas, an implementation of `start` might return `{Invoke,Send}`, telling the control object to do a local invocation and pass the marshalled invocation request so that it can be sent to other replicas.

The final step is to invoke `finish`, allowing the replication subobject to synchronize the replicas again (if needed). Again, the `finish` invocation is determined by the replication subobject, for which it returns `finish` after the invocation of `start` or `send`. Invoking `finish` generally returns `{Return}`, telling the control subobject that it can pass the method invocation's return value to the local client.

A distinctive feature of our model is that we allow method invocations at the semantics subobject to block on condition failures. For example, appending

Table 2. The control subobject's callback interface as used by the replication subobject.

Method	Description
`handle_request`	Called to invoke the specified method at the semantics subobject.
`getState`	Returns the (marshalled) state of the semantics subobject.
`setState`	Replaces current state of the semantics subobject with the state passed as an argument.

data to a bounded buffer might fail when the buffer fills—the replication subobject controls concurrent access to the semantics subobject. After invoking a method at the semantics subobject, the control object always calls `invoked`, informing the replication subobject whether a condition failure occurred, and passing control back to the replication subobject. If necessary, the current thread blocks inside the replication subobject. The replication subobject can then allow other invocations to take place, which might possibly change the state of the semantics subobject such that the blocked thread can continue successively.

Control subobject

The control subobject always invokes the semantics subobject's method. This subobject controls two types of invocation requests: those coming from the local client and those coming in through the network. The control subobject is also responsible for (un)marshalling invocation requests passed between itself and the replication subobject. The control subobject's interface offered to the local client is the same as the semantics subobject's (user-defined) interface. In addition, it offers the callback interface to the replication subobject shown in Table 2.

In general, when a local client invokes a method at the control subobject, the latter will eventually invoke that method at its local copy of the semantics subobject with permission from the replication subobject. Remote invocation requests—that is, requests that replication subobjects pass in remote address spaces—are eventually passed to the control subobject through `handle_request`. The control subobject then simply does the local invocation at the semantics subobject.

Communication subobject

The communication subobject, finally, is responsible for handling communication between parts of the distributed object that reside in different address spaces. It is generally a system-provided local object. Depending on what is needed from the other components, the communication subobject offers reliable or best-effort communication, connection-oriented or connectionless communication, and point-to-point or multicast facilities. Like the replication subobject, it offers a standard interface but allows many different implementations of that interface. The most important methods are those for sending and receiving messages, as well as methods to support request and reply semantics.

The main role of our communication subobjects is that they provide a uniform interface to underlying networks and operating systems concerning their communication facilities. By providing a standard interface, we can develop other local objects in a platform-independent way. Communication and replication subobjects are often unaware of the methods and state of the semantics subobject. This independence allows us to define standard interfaces for all replication subobjects and communication subobjects. Consequently, we can implement different policies but keep the interfaces the same. This also means that we can now easily adopt a policy by choosing an appropriate implementation from a library of class objects, which contain the implementation of subobjects—we can then dynamically download that implementation into our local object framework.

Process-to-object binding

To communicate through a distributed object, it is necessary for a process to first bind to that object. The result of binding is that the process can directly invoke the object's methods. In other words, a local object implementing a distributed object's interface is placed in the requesting process's address space. Binding itself consists roughly of two distinct phases: finding the distributed object and installing a local object. Figure 3 illustrates this. Finding a distributed object is separated into name and location look-up steps; installing the local object requires that we select a suitable contact address, as well as an implementation for that interface.

Finding a distributed object

To find a distributed object, a process must pass the object's name to a naming service. The naming service returns an object handle, which is a location-inde-

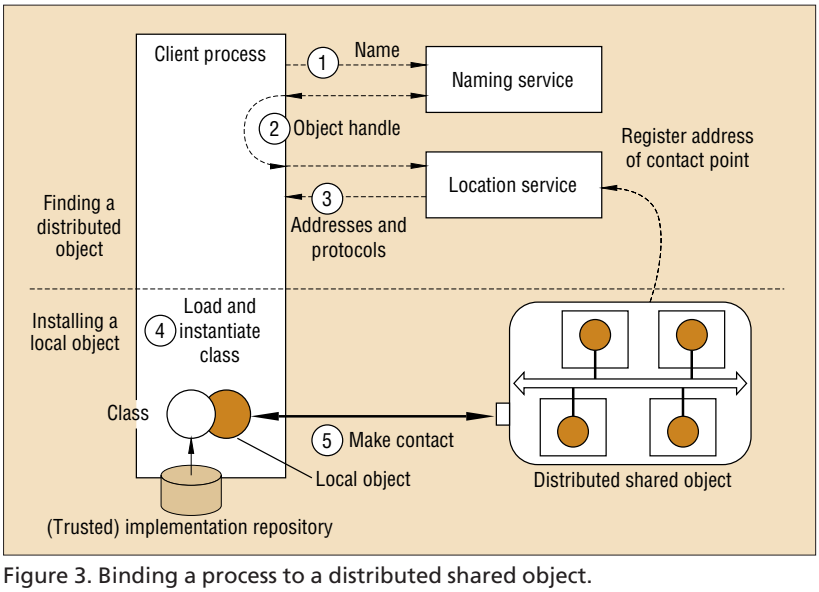

Figure 3. Binding a process to a distributed shared object.

pendent and universally unique object identifier, such as a 128-bit number (which is used to locate objects). It can be passed freely between processes as an object reference. It never changes over time and is guaranteed to refer to the same object even years later (if the object still exists). The object handle is given to a location service that returns one or more contact addresses. Globe thus uses a two-level naming hierarchy.

This organization allows us to separate issues related to naming objects from those related to contacting objects. In particular, it is now easy to support multiple and independent object names. Because an object handle does not change once it has been assigned to an object, a user can easily bind a private or locally shared name to an object without ever having to worry that the name-to-object binding will change without notice. On the other hand, an object can update its contact addresses at a location service without having to consider under which name its clients access it.

We can now remove all location information from names, thus making it easier to realize distribution transparency. However, we do require a scalable location service that can handle frequent contact-address updates in an efficient manner. We have designed such a service and are currently implementing a prototype version to test on the Internet. [8]

Installing a local object

Once a process knows where it can contact the distributed object, it needs to select a suitable address from the ones the location service returns. A contact address might be selected for its locality, but other criteria might determine preference of one address over another. For example, some addresses might belong to difficult-to-reach subnets or to subnets to which only low-bandwidth connections can be established. Other quality-of-service aspects might need to be considered as well. Note that an address selection service is a local service that builds its own administration concerning contact-address quality.

A contact address describes where and how the requested object can be reached. The latter is expressed as a protocol identifier. It specifies a complete stack of protocols that should be implemented on the client's side to communicate with the distributed object.

Of course, implementation selection might fail if a (trusted) implementation cannot be found. In that case, the binding process returns to the address-selection step, where the next best address is considered.

Current status

We built an initial prototype implementation of our system, concentrating on the support for distributed shared objects. Our initial prototype was implemented in ANSI C. We are currently developing a Java-based implementation.

Interfaces are written in an interface definition language (IDL). The prototype has an interface compiler that creates a C header file for each interface definition. The interface compiler also generates skeletons for (class) object implementations. A skeleton provides the necessary glue to turn a method invocation on a (local) object into a C function call. The programmer only has to implement one C function for each method.

The interface compiler also generates composite objects. A composite object encapsulates a collection of subobjects and allows them to be treated as a single entity. For example, our local object is constructed as a composition consisting of the four subobjects shown in Figure 2. These four subobjects are written manually with the help of generated skeletons. Generally, replication and communication subobjects are selected from a collection of subobjects supplied with the prototype.

A class object (containing the implementation of a subobject) is stored on the local file system and loaded at runtime into a process's address space. A configuration file specifies for each class name the file in which the corresponding class object is stored.

Object repositories support persistent distributed shared objects. An object repository provides a distributed object with support for storing its state persistently (on disk). It can activate passivated objects—that is, objects removed from address spaces. An object repository also provides a factory object: a distributed object that creates new persistent objects. An object repository is a simple Unix process that stores the state of the object it manages in a Unix file system. In the prototype, each factory creates only one type of object. During an object repository's configuration, it specifies what object types it can create.

An application uses a distributed object by binding to it. The prototype provides simple (Unix-style) programs that create and delete distributed objects, list the contents of directories, write and read objects, and so on. Our largest application so far is a Web proxy that converts HTTP requests into method invocations on distributed shared objects. To bind to an object, applications use both location and name services. The location service is implemented as a simple, centralized database. The name service is constructed as a collection of distributed directory objects.

Initial prototyping experiences

To allow concurrent access to objects, Globe supports multithreading. However, it is well known that correctly programming multithreaded applications is difficult. To minimize problems, we follow an approach in which two types of threads are strictly separated. Pop-up threads, which are used to handle requests coming in through the network, are allowed to invoke only methods from callback interfaces (except for semantics subobject methods). Likewise, threads originating from the local client can invoke only methods from regular interfaces. Furthermore, subobjects are programmed in such a way that critical regions need never be locked while a call is being made to another subobject.

As we explained, we have developed and implemented an IDL. Our IDL is similar to, for example, the CORBA IDL, except that we can also support interface specifications for local objects. Interfaces in C and Java are generated from IDL descriptions. This approach has proven to be highly effective, leading to well-designed subobjects. Nevertheless, the

control subobject currently has to be made by hand, which unnecessarily complicates object construction. It is better to specify the semantics subobject in an *object definition language*, from which, together with IDL descriptions, we can generate the control subobject. We are currently developing an ODL for Globe.

Being able to implement policies on a per-object basis proved to be highly effective. For example, because we were initially not interested in persistence, we used a single database to store the state of different distributed shared objects. The problem with this approach, which is basically the same as the one followed in CORBA, was that too many policy decisions had to be implemented outside the control of the object being stored. Later, we decided to follow the Globe paradigm more closely (in which each object is in full control of handling its own state). In our current prototype, each object implements its own persistence facilities, as well as the policy that goes along with it. This approach has turned out to be much more flexible and easier to implement and maintain.

The performance of our prototype, which is currently dominated by the time it takes for a process to bind to an object, confirmed that the granularity of distributed shared objects should be relatively large. For wide-area objects, network speed and delay will additionally determine performance, while the semantics subobject's size determines granularity. Unfortunately, in our model, a replication strategy operates on the entire state as contained in this subobject. This approach is not always appropriate. For example, when a semantics subobject is built from a number of Web pages, including icons, images, text, and graphics, we would like to apply different strategies for different parts of the subobject. Developing each part as a separate distributed shared object has an unacceptable performance penalty. We are currently investigating how we can support composite semantics subobjects whose elements can have separate replication strategies.

A Java-based prototype for the Web

Based on our first prototyping experiences, we are currently developing an implementation of Globe tailored to support scalable Web documents.[9] A Globe Web document is a collection of logically related Web pages. A page might consist of text, icons, images, sounds, and animations, as well as applets, scripts, and other forms of executable code. Each Globe Web document is constructed as a distributed shared object.

Instead of using C, we chose Java as our implementation language. Constructing a Globe Web document proceeds as follows. The elements that constitute the document (such as text, icons, applets, and graphics) are grouped together into what is called a *state archive*. As its name suggests, a state archive contains the state of the semantics subobject.

A semantics subobject offers a standard interface. For example, it is possible to add, remove, or replace elements. At present, each element is represented as a byte image and has an associated MIME type. Besides a standard interface for a semantics subobject, we offer a standard implementation of a control subobject and implementations for the interfaces of the replication and communication subobjects. These implementations jointly constitute a template for a Globe Web document's local object.

Finally, a developer has to choose Java classes that implement the replication and communication subobjects' interfaces. This leads to one or more *class archives*. Basically, a class archive contains a Java implementation of a specific replication strategy. The state archive, local object template, and a class archive are then grouped together into a single file from which a local object can be instantiated. If no suitable class is available, the implementer is free to write a new one.

To integrate our documents into the current Web infrastructure, we use a filtering gateway that communicates with standard Web clients (browsers), as shown in Figure 4. The gateway is a proxy that runs on a local server machine and accepts regular HTTP requests for a document. In our model, Globe Web documents are distinguished from other Web resources through naming. A Globe name is written as a URL with Globe as a scheme identifier. So, for example, *globe://cs.vu.nl/~steen/globe/* could be the name of our project's home document, constructed as a distributed shared object.

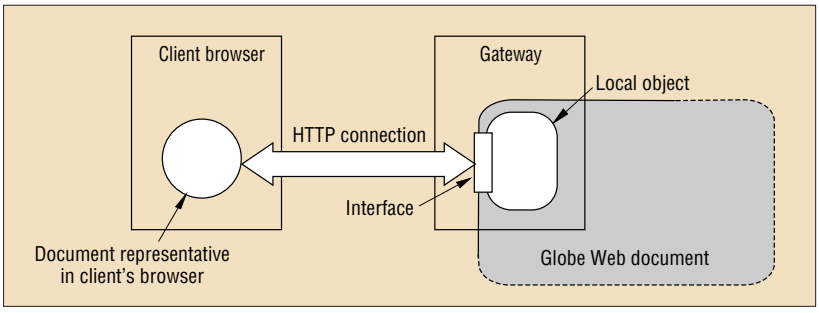

Figure 4. The general organization for integrating Globe Web services into the current Web.

The gateway accepts all URLs. Normal URLs are simply passed to existing (proxy) servers, whereas Globe URLs are used to actually bind to the named distributed shared object. Unfortunately, existing browsers cannot handle Globe names, which is why we embed these names in URLs with HTTP as scheme identifiers. In addition, we use Java applets to support interactive documents. We are investigating the use of browser plug-ins to allow browser extensions for support of Globe's distributed shared objects.

WE HAVE FINISHED THE initial architectural design of our system, leaving a number of subjects open for further research. For example, we are currently designing a security architecture. Furthermore, we are concentrating on specific schemes for wide-area replication and persistence, mechanisms that support large-scale applications composed of many distributed objects, and persistence. Our efforts concentrate on developing a Java-based implementation for constructing scalable Web documents.

ACKNOWLEDGMENTS

Many ideas presented in this article have been formed during discussions with other participants in the Globe project: Arno Bakker, Gerco Ballintijn, Franz Hauck, Anne-Marie Kermarrec, Ihor Kuz, and Patrick Verkaik. We gratefully acknowledge their contributions. Leendert van Doorn is acknowledged for his contributions to the detailed design and implementation of local objects. Finally, we thank Henri Bal, Koen Langendoen, and the anonymous referees for their valuable assistance in improving this article.

References

1. *DCOM Technical Overview*, Microsoft Corp., Redmond, Wash., 1996.
2. W. Rosenberry, D. Kenney, and G. Fisher, *Understanding DCE*, O'Reilly and Associates, Sebastopol, Calif., 1992.
3. *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, OMG Document 96.03.04, Object Management Group, Framingham, Mass., 1998.
4. M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *Computer*, Vol. 23, No. 5, May 1990, pp. 9–21.
5. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1994.
6. N. Islam, "Customizing System Software Using OO Frameworks," *Computer*, Vol. 30, No. 2, Feb. 1997, pp. 69–78.
7. B. Neuman, "Scale in Distributed Systems," *Readings in Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 463–489.
8. M. van Steen et al., "Locating Objects in Wide-Area Systems," *IEEE Communication Magazine*, Vol. 36, No. 1, Jan. 1998, pp. 104–109.
9. M. van Steen et al., "A Scalable Middleware Solution for Advanced Wide-Area Web Services," *Middleware '98*, Springer Verlag, London, 1998, pp. 37–53.

Maarten van Steen is an assistant professor at the Vrije Universiteit in Amsterdam. He has worked at an industrial research laboratory for several years in the field of parallel programming environments. His research interests include operating systems, computer networks, distributed systems, and distributed-software engineering. He received an MSc in applied mathematics from Twente University and a PhD in computer science from Leiden University. He is a member of the IEEE Computer Society and ACM. Contact him at Vrije Univ., Dept. of Mathematics and Computer Science, De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands; steen@cs.vu.nl.

Philip Homburg is a PhD student working on Globe's overall design. He graduated from the Vrije Universiteit in Amsterdam. Before starting with his PhD studies, he wrote software to transparently connect multiple Amoeba sites over the Internet as part of the Starfish project. Contact him at Vrije Univ., Dept. of Mathematics and Computer Science, De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands; philip@cs.vu.nl.

Andrew S. Tanenbaum is a professor of computer science at the Vrije Universiteit in Amsterdam and dean of an interuniversity computer science graduate school called ASCI. He is the principal designer of three operating systems: TSS-11, Amoeba, and Minix. He was also the chief designer of the Amsterdam Compiler Kit. He has a BS from MIT and a PhD from the University of California at Berkeley. He is a Fellow of the ACM, a Fellow of the IEEE, and a member of the Royal Dutch Academy of Sciences. In 1994, he was the recipient of the ACM Karl V. Karlstrom Outstanding Educator Award, and in 1997, he won the SIGCSE award for contributions to computer science. Contact him at Vrije Univ., Dept. of Mathematics and Computer Science, De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands; ast@cs.vu.nl.